**LCG Generator**

LCG generator is defined by [recurrence relation](https://en.wikipedia.org/wiki/Recurrence_relation):

{\displaystyle X\_{n+1}=\left(aX\_{n}+c\right)~~{\bmod {~}}~m}Xn+1 = (aXn + c ) Mod(m)

Here a is a constant multiplication factor, c is addition factor and m is modulus.

m>0, 0<a<m, 0<=c<m, 0<=x0<m

If c= 0 then it is pure multiplicative congruential generator and known as *mixed congruential generator* and it has full period of (m-1 ) and can be subjected to spectral test.

For the time period length of the sequence of random numbers = m

* c and m must be relatively prime or co-prime (their common divisor is 1 ) or gcd(c,m)=1
* a-1 should be divisible by all prime factors of m
* a-1 should divisible by 4 if m is divisible by 4.

Typically, LCGs are more useful when uniformly distributed on the interval [0; 1], this way the quality of output is comparable with other LCGs with different set size and other random number generators in general. This is done by dividing the output of a LCG by m.

Psuedocode:

LCG <- function(n,m,a,c,X0) {

X <- c()

Xn <- X0

for(i in 1:n){

Xn <- (a\*Xn + c) %% m

X[i] <- Xn

}

X <- X/m

return(X)

It can pass formal test of randomness

**Disadvantages**

* LCG is vulnerable to the cyber attack if used to generate keys in cryptorism  because it is possible to recover the parameters of LCGs in polynomial time given several observed outputs

**Blum Blum Shub Generator**

It generate the pseudo random generators via following recurence relation:

Xn+1= X^2ModM

* Output is the least significant bit of Xn+1 or parity bit of Xn+1.
* M is product of two larger prime numbers that should be congruent to 3Mod4 ( If a is congruent to bModc then a-b should divisible by c).
* Seed X0 should co-prime with M and should not be 1 or 0.

**Disadvantages**

However, BBS is not a permutation generator as the RNGs mentioned above. A $k$-permutation generator is a program which produces permutations of a set of $k$ distinct objects. Each of these permutations is called a $k$-permutation. A full-period RNG with period length $n$ is also an $n$-permutation generator since any $n$ consecutive outputs form an $n$-permutation. The period length of BBS is much lesses than $M$, therefore, it can not be adapted to produce uniform random variates between 1 and $M-1$. For example, use use $M=16873$, and hence the period is much lesser than 16873. Due to this reason, BBS failed all of the statistical tests we have conducted and is therefore not suitable for stochastic simulations.

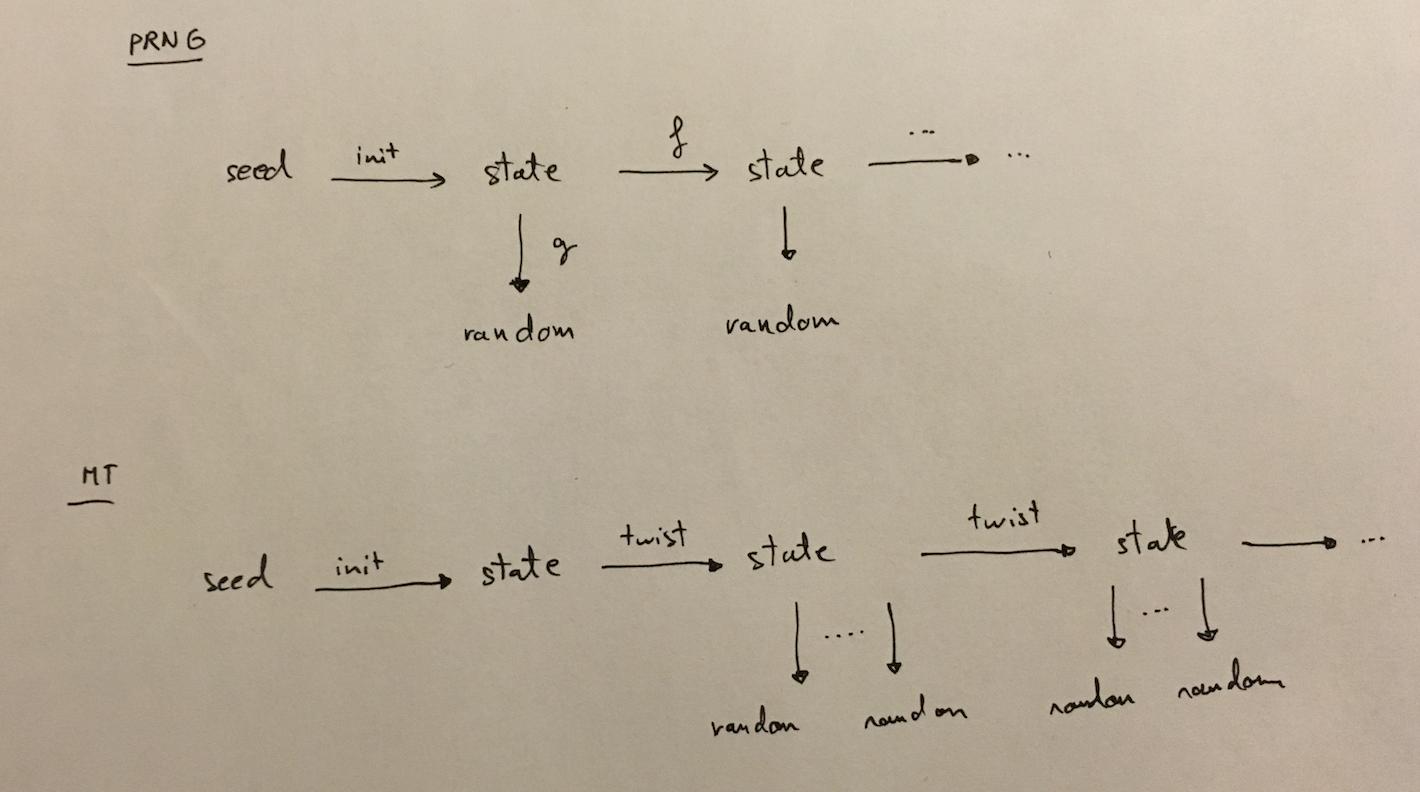
There is a proof reducing its security to the [computational difficulty](https://en.wikipedia.org/wiki/Computational_complexity_theory) of solving the [quadratic residuosity problem](https://en.wikipedia.org/wiki/Quadratic_residuosity_problem).[[1]](https://en.wikipedia.org/wiki/Blum_Blum_Shub#cite_note-blum1986-1) When the primes are chosen appropriately, and [O](https://en.wikipedia.org/wiki/Big_O_notation)([log](https://en.wikipedia.org/wiki/Logarithm) log *M*) lower-order bits of each *xn* are output, then in the limit as *M* grows large, distinguishing the output bits from random should be at least as difficult as solving the Quadratic residuosity problem modulo *M*.

**Example-**

If p=11, q=19, X0=3 then M= 209

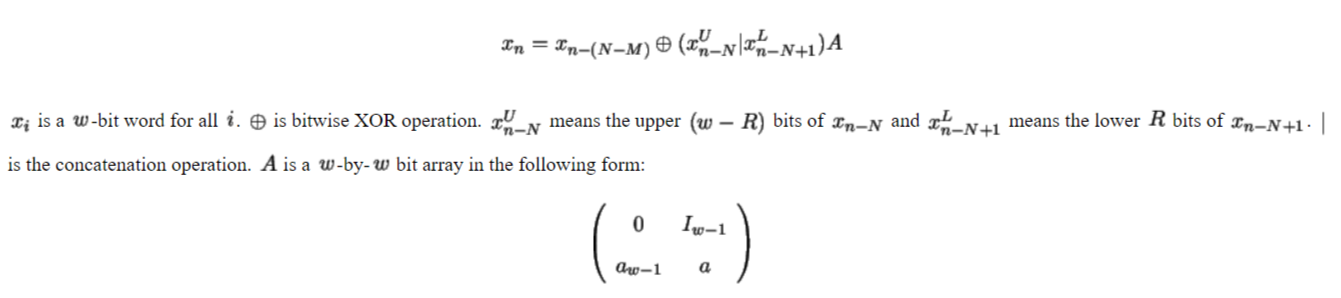
Sequence will 9, 81, 82, 36, 42, 92.

**Mersenne Twister Algorithm**



Its first state doesn’t produce any random numbers. The next states after initial state produce long sequence of 264 random numbers. (The state is transformed into random numbers via a g function ).

It has long period of 2^19937-1 and based on the following linear recurrence:



Here Iw-1 is the (w-1) by (w-1) identity matrix and $a = (a_{w-2}, a_{w-3}, \ldots, a_0)$ The parameters of MT are N=624, M=397, R=31, w = 32 and a= 9908B0DF in hexadecimal.

**int**[0..**n**-1] MT

**int** index := **n**+1

**const int** lower\_mask = (1 << **r**) - 1 *// That is, the binary number of* ***r*** *1's*

**const int** upper\_mask = lowest **w** bits of ([**not**](https://en.wikipedia.org/wiki/Bitwise_operation#NOT) lower\_mask)

*// Initialize the generator from a seed*

**function** seed\_mt(**int** seed) {

index := **n**

MT[0] := seed

**for** i **from** 1 **to** (**n** - 1) { *// loop over each element*

MT[i] := lowest **w** bits of (**f** \* (MT[i-1] [**xor**](https://en.wikipedia.org/wiki/Bitwise_operation#XOR) (MT[i-1] >> (**w**-2))) + i)

}

}

*// Extract a tempered value based on MT[index]*

*// calling twist() every* ***n*** *numbers*

**function** extract\_number() {

**if** index >= **n** {

**if** index > **n** {

**error** "Generator was never seeded"

*// Alternatively, seed with constant value; 5489 is used in reference C code*[[46]](https://en.wikipedia.org/wiki/Mersenne_Twister#cite_note-46)

}

twist()

}

**int** y := MT[index]

y := y [**xor**](https://en.wikipedia.org/wiki/Bitwise_operation#XOR) ((y >> **u**) [**and**](https://en.wikipedia.org/wiki/Bitwise_operation#AND) **d**)

y := y [**xor**](https://en.wikipedia.org/wiki/Bitwise_operation#XOR) ((y << **s**) [**and**](https://en.wikipedia.org/wiki/Bitwise_operation#AND) **b**)

y := y [**xor**](https://en.wikipedia.org/wiki/Bitwise_operation#XOR) ((y << **t**) [**and**](https://en.wikipedia.org/wiki/Bitwise_operation#AND) **c**)

y := y [**xor**](https://en.wikipedia.org/wiki/Bitwise_operation#XOR) (y >> **l**)

index := index + 1

**return** lowest **w** bits of (y)

}

*// Generate the next* n *values from the series x\_i*

**function** twist() {

**for** i **from** 0 **to** (**n**-1) {

**int** x := (MT[i] [**and**](https://en.wikipedia.org/wiki/Bitwise_operation#AND) upper\_mask)

+ (MT[(i+1) [**mod**](https://en.wikipedia.org/wiki/Modulo_operation) **n**] [**and**](https://en.wikipedia.org/wiki/Bitwise_operation#AND) lower\_mask)

**int** xA := x >> 1

**if** (x [**mod**](https://en.wikipedia.org/wiki/Modulo_operation) 2) != 0 { *// lowest bit of x is 1*

xA := xA [**xor**](https://en.wikipedia.org/wiki/Bitwise_operation#XOR) **a**

}

MT[i] := MT[(i + **m**) [**mod**](https://en.wikipedia.org/wiki/Modulo_operation) **n**] [**xor**](https://en.wikipedia.org/wiki/Bitwise_operation#XOR) xA

}

index := 0

}

Ransac Algorithm

**Random sample consensus** (**RANSAC**) is an [iterative method](https://en.wikipedia.org/wiki/Iterative_method) to estimate parameters of a mathematical model from a set of observed data that contains [outliers](https://en.wikipedia.org/wiki/Outliers), when outliers are to be accorded no influence on the values of the estimates. Therefore, it also can be interpreted as an outlier detection method.[[1]](https://en.wikipedia.org/wiki/Random_sample_consensus#cite_note-1) It is a non-deterministic algorithm in the sense that it produces a reasonable result only with a certain probability, with this probability increasing as more iterations are allowed. The algorithm was first published by Fischler and Bolles at [SRI International](https://en.wikipedia.org/wiki/SRI_International) in 1981. They used RANSAC to solve the Location Determination Problem (LDP), where the goal is to determine the points in the space that project onto an image into a set of landmarks with known locations.

A basic assumption is that the data consists of "inliers", i.e., data whose distribution can be explained by some set of model parameters, though may be subject to noise, and "outliers" which are data that do not fit the model. The outliers can come, for example, from extreme values of the noise or from erroneous measurements or incorrect hypotheses about the interpretation of data. RANSAC also assumes that, given a (usually small) set of inliers, there exists a procedure which can estimate the parameters of a model that optimally explains or fits this data.